CMSC 39600: PCPs, codes and inapproximability Oct 2, 2007

Lecture 3: Linearity Testing

Lecturer: Prahladh Harsha *Scribe: Joshua A. Grochow*

In today's lecture we'll go over some preliminaries to the coding theory we will need throughout the course. In particular, we will go over the Walsh-Hadamard code and the Blum-Luby-Rubinfeld linearity test. In the following lecture we will use these tools to show that $NP \subseteq PCP[\text{poly}, O(1)]$ (recall that our goal is ultimately to bring the polynomial amount of randomness down to a logarithmic amount). A common feature throughout the course is that we will use codes to construct PCPs and vice-versa, construct useful codes from PCPs.

3.1 Coding Theory – Preliminaries

Coding theory is the study of how to communicate reliably over a noisy channel. The most common setting is as follows: A message m is put through an encoder E, yielding a value $E(m)$, also called the codeword (typically much longer than m). The codeword $E(m)$ is then sent through the noisy channel and arrives at the other end with some noise introduced η by the channel as $E(m) + \eta$. The decoder D then takes $E(m) + \eta$ as input, and ideally outputs m as long as the noise is not too much. (Note that it is expected of the decoder that when applied to $E(m)$ it outputs m.)

Formally, a code is specified by an *encoding function* $C : \{0,1\}^k \to \{0,1\}^n$; the outputs of C are called *codewords*. The *rate* of the code C is k/n, i.e. the ratio of input bits to codeword bits. Heuristically, this is the amount of information of the input message per bit of the codeword.

An important notion in coding theory is that of the distance between two codewords; here (and typically) we use either the Hamming distance $\Delta(x, y) = \#\{i | x_i \neq y_i\}$ or the normalized (Hamming) distance $\delta(x, y) = \Delta(x, y)/n$ (where $n = |x| = |y|$).

The *distance d of a code C* is the minimum distance between two distinct codewords, i.e., $\min_{x \neq y}\{\Delta(C(x), C(y))\}$. For a code of distance d, if the number of bits flipped is strictly less than $d/2$ (i.e., weight$(\eta) \leq d/2$), then $E(m) + \eta$ can be uniquely decoded to m. A code $C : \{0,1\}^k \to \{0,1\}^n$ with distance d is called an $(n, k, d)_2$-code, where n is the size of the codewords, k the size of the input, d the distance of the code, and 2 the size of the alphabet $\{0,1\}$.

A code C is called *linear* if for all x and y, if x and y are codewords then so is $x + y$ (where addition is performed bitwise modulo 2 – i.e. XOR)[1]. To indicate that a $(n, k, d)_2$-code is also linear, we use the notation $[n, k, d]_2$-code (with square brackets). The word "code," is commonly used to refer to both the encoding function C (as stated above), as well as the set of all codewords $\text{Im}(C)$.

Typically, the most significant trade-off in coding theory is that between the rate and distance of a code. For example, given a particular rate, we might ask what is the best

[1]Note that if we work over a larger alphabet than binary – e.g. over a larger finite field, we require the additional constraint that x is a codeword then so are all scalar multiples of x (i.e., αx)

distance that can be achieved.

3.1.1 Algorithmic Questions and Sublinear Time Algorithms

There are three main algorithmic questions that arise in coding theory:

1. Complexity of encoding;
2. Error detection: given $r \in \{0,1\}^n$, decide if r is a codeword; and
3. Error correction: given $r \in \{0,1\}^n$, find $x \in \{0,1\}^k$ minimizing $\Delta(r, C(x))$.

In this course, we will interested in these questions in the context of *sublinear time algorithms*. We need to be specific what we mean by sublinear time. Note that a sublinear time algorithm A to compute a function f $\{0,1\}^k \to \{0,1\}^n$ doesn't have enough time to read it's input or write it's output. We get over this by accessing the input and writing the output by oracle access. That is, an oracle machine A is a sublinear time algorithm for f if $A^x(j) = f(x)_j$ where $f(x)_j$ is the j-th bit of $f(x)$. Note that j need only be $\log n$ bits, so can be read entirely in sublinear time. More formally,

- The input is represented implicitly by an oracle. Whenever the sublinear time algorithm wants to access the j^{th} bit of the input string x (for some $j \in [k]$), it queries the input x-oracle for the j^{th} bit and obtains x_j.

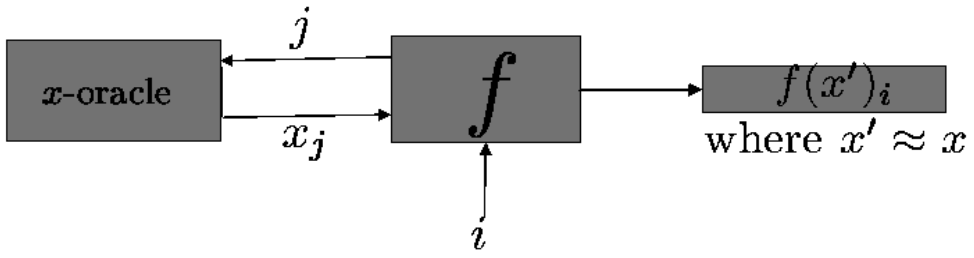

Figure 1: Sublinear Time Algorithms

- The output is not explicitly written by the algorithm, instead it is only implicitly given by the algorithm. Formally, on being queried for index i of the output string $f(x)$ (for some $i \in [n]$), the algorithm outputs the bit $f(x)_i$. Thus, the algorithm itself behaves as an oracle for the string $f(x)$, which in turn has oracle access to the input oracle x.

- Since the algorithm does not read the entire input x, we cannot expect it compute the output $f(x)$ exactly. We instead relax our guarantee on the output as follows: On input $x \in \{0,1\}^k$, the algorithm must compute $f(x')$ exactly for some $x' \in \{0,1\}^k$ that is ϵ-close to the actual input x. In other words, the algorithm computes functions on some approximation to the input instead of the input itself.

Property testing, for those familiar with it, is typically done in this framework. Figure 1 gives a pictorial description of a sublinear time algorithm with the above mentioned relaxations.

Now we'll consider the algorithmic questions of coding theory in the sublinear time framework. Let's consider the questions above one by one:

1. For a code to have good error-correcting properties, most bits of the codeword needs to depend on most message-bits. Taking this into account, it does not seem reasonable to expect a "good" code to have sublinear time encoding. (However, there has been some recent work in the area of locally encodable codes by relaxing some of the error-correction properties of the code).

2. Error detection: in this context, error detection is known as *local testing*. That is, given $r \in \{0,1\}^n$, test if either r is a codeword or r is far from every codeword (similar to the gap problems we saw earlier, and also to property testing).

3. Error correction is known as *local decoding* in this context. Given j, the decoder queries some bits of a noisy codeword $C(x) + \eta$ and outputs the j-th bit of x. more formally, we say a code is *locally decodable* if there exists a local decoder Dec such that for all x and r where $\Delta(r, C(x)) < \epsilon$, the decoder satisfies

$$\forall i \in \{1, \ldots, k\}, \Pr[\mathsf{Dec}^r(i) = x_i] \geq \frac{3}{4}.$$

   If there is such a decoder for a given code C and it makes fewer than q queries, then we say C is (q, ϵ)-locally decodable.

Obviously, sublinear time algorithms in general, and local decoding and local testing algorithms in particular, will be randomized algorithms.

Note that local decoding is only required to work when the input is sufficiently close to a codeword, whereas local testability determines whether a given string is close to a codeword or far from any codeword. Thus the local decodability of a code says nothing about its local testability.

3.2 The Walsh-Hadamard Code and Linearity Testing

Now onto our first code, the Walsh-Hadamard code. This will be the main tool in proving that NP has exponential size PCPs, i.e. $NP \subseteq PCP[\text{poly}, O(1)]$. There are two dual views of the Walsh-Hadamard code: on the one hand, $WH(x)$ is the evaluation of every linear function at x; on the other hand, it consists of the dot product of x with every $\ell \in \{0,1\}^n$.

A function $f\ \{0,1\}^k \to \{0,1\}$ is *linear* if $\forall x, y, f(x+y) = f(x) + f(y)$.

For example, for any $a \in \{0,1\}^k$, the function $\ell_a(x) = \sum a_i x_i \mod 2$ (i.e. the dot product of a and x as vectors over $GF(2)$) is a linear function. In fact, $\mathcal{L}_k = \{\ell_a | a \in \{0,1\}^k\}$ is the set of *all* linear boolean functions. (Proof: observe that the space of linear functions is a vector space, and has the same dimension as $\mathcal{L}_k$.)

The Walsh-Hadamard code $WH : \{0,1\}^k \to \{0,1\}^{2^k}$ is then defined as $WH(x) = \ell_x$, i.e. $WH(x)$ is the truth table of ℓ_x. More formally, for any $a \in \{0,1\}^k$, the a-th bit of the Walsh-Hadamard codeword $WH(x)$ is $WH(x)_a = \ell_x(a)$.

Since any two distinct linear functions disagree on exactly half the set of inputs (i.e, $\Pr_a[\ell_x(a) \neq \ell_y(a)] = 1/2$, for $x \neq y$), the fractional distance of the Walsh-Hadamard codes is $1/2$. Thus the Walsh-Hadamard code is a $[k, 2^k, 2^{k-1}]$-code. It has very good distance, but poor rate.

It is useful to note that there are two dual views of the Walsh-Hadamard code based on the fact that $WH(x)_a = \ell_x(a) = \ell_a(x)$. Thus, $WH(x)$ is both the evaluation of the linear function ℓ_x at every point as well as the evaluation of every linear function at the point x.

Note that the WH code has the special property that the input bits x_i are in fact a subset of the codeword bits, since $x_i = \ell_x(e_i)$. Codes with this property are called *systematic codes.*

3.2.1 Local Decodability of the Walsh-Hadamard Code

Decoding the Walsh-Hadamard code is very simple. Given a garbled codeword $f : \{0,1\}^k \to \{0,1\}$, which is δ-close to some Walsh-Hadamard codeword, the decoder Dec works as follows: (The decoder Dec has oracle access to the function f)

Dec^f : "On input z,

1. Choose $r \in_R \{0,1\}^k$
2. Query $f(z+r)$ and $f(r)$
3. Output $f(z+r) - f(r)$ "

Claim 3.1. *If $f : \{0,1\}^k \to \{0,1\}$ is δ-close to $WH(x)$, then,*

$$\Pr[\mathsf{Dec}^f(z) = \ell_x(z)] \geq 1 - 2\delta, \forall x \in \{0,1\}^k.$$

Proof. Since f is δ-close to ℓ_x, we have that for a random r, the probability that $f(z+r) = \ell_x(z+r)$ is at least $1-\delta$, and so is the probability that $f(r) = \ell_x(r)$. If both of these conditions hold, then $\mathsf{Dec}^f(z) = \ell_x(z)$, by the linearity of ℓ_x. Thus $\Pr_r[\mathsf{Dec}^f(z) = \ell_x(z)] \geq 1 - 2\delta$. □

Since the decoder only makes two queries, the Walsh-Hadamard code is 2-locally decodable.

3.2.2 Local Testability of the Walsh-Hadamard Code: Linearity Testing

To locally test the WH code, we wish to test whether a given truth table is the truth table of a linear function. The problem of local testing of the WH code is more commonly called *linearity testing*. Formally, given f $\{0,1\}^k \to \{0,1\}$ we want to test whether it is a linear function (WH codeword) or far from linear.

The test is, as in the WH decoder, quite simple: pick y and z uniformly at random from $\{0,1\}^k$ and check that $f(z) + f(y) = f(z+y)$. This test was proposed and first analyzed by Blum, Luby and Rubinfeld [BLR93].

$\mathsf{BLR\text{-}Test}^f$: " 1. Choose $y, z \in_R \{0,1\}^k$ independently

2. Query $f(y), f(z),$ and $f(y+z)$
3. Accept if $f(y) + f(z) = f(y+z)$. "

Obviously if f is linear, this test will pass with probability 1. The question is, can there be function that are far from linear but still pass this test with high probability? No, as shown in the following

Theorem 3.2. *If f is δ-far from linear, then*

$$\Pr[\mathsf{BLR\text{-}Test}^f \text{accepts }] \leq 1 - \delta.$$

The above theorem is tight since a random function is 1/2-far from linear, and passes the BLR-Test with probability exactly 1/2.

The original proof of Blum, Luby and Rubinfeld [BLR93] is a combinatorial proof of a weaker version of the above theorem, but we will give an algebraic proof, as similar techniques will arise later in the course. This algebraic proof is due to Bellare, Coppersmith, håstad, Kiwi and Sudan [BCH+96]. Before proceeding to the proof, we will first need to equip ourselves with some basics of Boolean Fourier analysis.

3.2.3 Fourier Analysis

First, rather than working over $\{0, 1\}$ as the output of a linear function, it will be convenient to treat the output space as $\{+1, -1\}$ (the roots of unity) under multiplication. Thus the Boolean 0 corresponds to 1 in this setting, and the Boolean 1 corresponds to -1. Linearity now takes the form: $f\ \{0, 1\}^n \to \{+1, -1\}$ is linear if $f(x + y) = f(x)f(y)$.

Consider the family of functions $\mathcal{F} = \{f : \{0, 1\}^k \to \mathbb{R}\}$ and equip $\mathcal{F}$ with an addition $(f + g)(x) = f(x) + g(x)$. It is clear that $\mathcal{F}$ is a vector space over the reals. Furthermore, the characteristic functions $\{\delta_a | a \in \{0, 1\}^k\}$ are a basis, where $\delta_a(x) = 1$ if $x = a$ and 0 otherwise. Thus $\mathcal{F}$ has dimension 2^k.

We will now show that the linear functions of the form $\chi_a(x) = (-1)^{\ell_a(x)} = (-1)^{\sum a_i x_i (\mod 2)}$ also form a basis for $\mathcal{F}$. For this, it is fruitful to define an inner product on $\mathcal{F}$ as follows:

$$\langle f, g\rangle = \mathrm{Exp}_{x\in\{0,1\}^k}\left[f(x)g(x)\right] = \frac{1}{2^k} \cdot \sum_{x\in\{0,1\}^k} f(x)g(x).$$

(It is an easy exercise to check that this is in fact an inner product.) Note that there are already $2^k = \dim \mathcal{F}$ functions of this form, so all we need to do is show that they are linearly independent.

We begin by examining a few basic properties of the functions χ_a. First, note that

Property 1. $\chi_a(x + y) = \chi_a(x)\chi_a(y)$

i.e. χ_a is linear, as mentioned previously. Second,

Property 2. $\chi_{a+b}(x) = \chi_{a-b}(x) = \chi_a(x)\chi_b(x)$

i.e. $\chi_a(x)$ is also linear in a; this should come as no surprise, because of the duality of $\ell_a(x)$ as both a linear function of a and of x.

The first property we will need that is not entirely obvious is that

Property 3.

$$\mathrm{Exp}_x[\chi_a(x)] = \begin{cases} 1 & \textit{if } a = 0 \\ 0 & \textit{otherwise} \end{cases}$$

Proof. If $a = 0$, this clearly holds. If $a \neq 0$, then by permuting the indices we may assume that $a_1 \neq 0$ without loss of generality. Then we have

$$\begin{aligned} 2^k \cdot \mathrm{Exp}_x[\chi_a(x)] &= \sum_x (-1)^{\sum a_i x_i} \\ &= \sum_{x:x_1=1} (-1)^{\sum a_i x_i} + \sum_{x:x_1=0} (-1)^{\sum a_i x_i} \end{aligned}$$

Then, since $(-1)^{\ell_a(0y)} = -(-1)^{\ell_a(1y)}$, these two sums exactly cancel out. □

We then have,

Property 4.

$$\langle \chi_a, \chi_b \rangle = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}.$$

This follows from the above via $\langle \chi_a, \chi_b \rangle = E_x[\chi_a(x)\chi_b(x)] = E_x[\chi_{a-b}(x)]$, and then applying the above fact.

Thus, the χ_a form not only a basis for $\mathcal{F}$, but also an orthonormal basis for $\mathcal{F}$. Since the χ_a form an orthonormal basis, for any $f \in \mathcal{F}$ we may write $f = \sum_a \hat{f}_a \chi_a$ for $\langle f, \chi_a \rangle = \hat{f}_a \in \mathbb{R}$. These $\hat{f}_a$ are called the *Fourier coefficients* of f.

Observer that if the normalized distance $\delta(f, \chi_a) = \epsilon$, then $\hat{f}_a = 1(1-\epsilon) + (-1)\epsilon = 1 - 2\epsilon$, so the Fourier coefficients capture the normalized distance from linear functions.

Now we come to one of the most basic useful facts in Fourier analysis:

Property 5 (Parseval's identity)**.** $\langle f, f \rangle = \sum \hat{f}_a^2$

Proof. Writing f in terms of the basis χ_a, we get:

$$\begin{aligned} \langle f, f \rangle &= \langle \sum_a \hat{f}_a \chi_a, \sum_b \hat{f}_b \chi_b \rangle \\ &= \sum_{a,b} \hat{f}_a \hat{f}_b \langle \chi_a, \chi_b \rangle \text{ (by linearity of } \langle \cdot, \cdot \rangle) \\ &= \sum_a \hat{f}_a^2 \end{aligned}$$

where the last line follows from the previous because the χ_a form an orthonormal basis. □

Corollary 3.3. *In particular, if f is a Boolean function, i.e. ± 1-valued, then $\sum \hat{f}_a^2 = 1$.*

3.2.4 Proof of Soundness of BLR-Test

Finally, we come to the proof of the soundness of the Blum-Luby-Rubinfeld linearity test:

Proof of Theorem 3.2. Suppose f is δ-far from any linear function. Note that we can rewrite the linearity condition $f(x)f(y) = f(x+y)$ as $f(x)f(y)f(x+y) = 1$, since f is ± 1-valued. Then

$$\Pr_{x,y}[\text{BLR-Test accepts } f] = \Pr_{x,y}[f(x)f(y)f(x+y) = 1]$$

Note that for any random variable Z with values in $\{+1,-1\}$, $Pr(Z=1) = \text{Exp}[\frac{1+Z}{2}]$, since if $Z=1$, then $(1+Z)/2 = 1$ and if $Z=-1$, then $(1+Z)/2 = 0$, so $(1+Z)/2$ is an indicator variable for the event $Z=1$. Thus we have

$$\begin{aligned}
\Pr_{x,y}[\textsf{BLR-Test} \text{ accepts } f] &= \text{Exp}_{x,y}\left[\frac{1+f(x)f(y)f(x+y)}{2}\right] \\
&= \frac{1}{2} + \frac{1}{2}\text{Exp}_{x,y}\left[f(x)f(y)f(x+y)\right]
\end{aligned}$$

Now, writing out f in terms of its Fourier coefficients, we get

$$\begin{aligned}
\Pr_{x,y}[\textsf{BLR-Test} \text{ accepts } f] &= \frac{1}{2} + \frac{1}{2}\text{Exp}_{x,y}\left[\sum_{a,b,c} \hat{f}_a\hat{f}_b\hat{f}_c\chi_a(x)\chi_b(y)\chi_c(x+y)\right] \\
&= \frac{1}{2} + \frac{1}{2}\text{Exp}_{x,y}\left[\sum_{a,b,c} \hat{f}_a\hat{f}_b\hat{f}_c\chi_a(x)\chi_b(y)\chi_c(x)\chi_c(y)\right]
\end{aligned}$$

Then, we apply linearity of expectation and the fact that x and y are independent to get:

$$\begin{aligned}
\Pr_{x,y}[\text{BLR accepts } f] &= \frac{1}{2} + \frac{1}{2}\sum_{a,b,c} \hat{f}_a\hat{f}_b\hat{f}_c\text{Exp}_x\left[\chi_a(x)\chi_c(x)\right]\text{Exp}_y\left[\chi_b(y)\chi_c(y)\right] \\
&= \frac{1}{2} + \frac{1}{2}\sum_{a,b,c} \hat{f}_a^3 \\
&\leq \frac{1}{2} + \frac{1}{2}\max_a(\hat{f}_a)\sum_a \hat{f}_a^2 \\
&= \frac{1}{2} + \frac{1}{2}\max_a(\hat{f}_a) \text{ (by Parseval's identity)} \\
&= 1-\delta
\end{aligned}$$

where the last line follows from the fact that f is δ-far from linear, so its largest Fourier coefficient can be at most $1-2\delta$, as noted previously. □

References

[BCH+96] Mihir Bellare, Don Coppersmith, Johan Håstad, Marcos A. Kiwi, and Madhu Sudan. *Linearity testing in characteristic two.* IEEE Transactions on Information Theory, 42(6):1781–1795, November 1996. (Preliminary version in 36*th FOCS*, 1995). doi:10.1109/18.556674.

[BLR93] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. *Self-testing/correcting with applications to numerical problems.* J. Computer and System Sciences, 47(3):549–595, December 1993. (Preliminary Version in *22nd STOC*, 1990). doi:10.1016/0022-0000(93)90044-W.